To read data from memory, CORE is used in an arithmetic expression with a subscript that indicates the location of memory to read. Information can be stored into memory by using CORE in an assignment statement.

```
print core (8192);   /* print contents of location 8192 */
x = core (i);
core (8192) = 0;          /* set location 8192 to zero */
core (ptr) = stack_top*3;
```

The specified location of memory is accessed as if it were a fixed point variable. If the program tries to read a location of memory that does not exist, the computer will halt and the program will stop running.

The LOCATION Function

The LOCATION function is used to reference absolute locations of memory during a procedure call. The keyword LOCATION (or the shortened form LOC) is followed by an expression, variable, or constant in parentheses. LOCATION can only appear in an actual parameter list during a procedure call. Observe the following example:

```
call readdata (0, 0, location (8192), 256);
```

When LOCATION is used, the computed expression is passed to the procedure as the start of an array. When using the LOCATION function the programmer must be sure to protect the locations of memory in which his program is residing. The memory map printed by the compiler in response to the COMPILE command can be used to determine this information, or refer to the appendix "Memory Layout" for information about determining free memory boundaries at runtime.

Interesting results can be obtained by using ADDR in conjunction with LOCATION. This is most often done when the user wishes to read data from the disk into element one of an array rather than element zero:

```
call readdata (0, 0, loc (addr (buf (1))), 256);
```

By using the LOCATION and ADDR functions, the user can implement dump and patch routines for referencing absolute locations of memory directly. Obviously, the user can only reference locations of memory that are unused by his program if system crashes are to be avoided.

Section VI - Procedures

The procedure definition facility of Scientific XPL allows modular programming, so that a program can be divided into logical sections. A well-designed procedure does a single application-related task (e.g., find the cosine of an angle, push an item onto the stack, or emulate a VT100 terminal). Furthermore, communication to that procedure should be by parameter passing only and the parameters should be restricted to application data (e.g., the angle to find the cosine of or the item to push onto the stack). Such procedures are conceptually simple, easy to debug and maintain, and easily incorporated into a large program. They can form a basis for a procedure library, if a family of similar programs is being developed.

A procedure is defined near the start of the program, and is then invoked (or called) from other parts of the program. During the procedure call, program control is transferred from the point of call to the top of the procedure, the procedure body is executed, and then program control is returned to the point of call.

A full discussion of XPL procedures is contained in this section, following an introductory section on XPL program structure.

Program Structure

Scientific XPL programs generally have a common structure, so that events happen in a certain order. One reason for this structure is the basic rules of the language, such as XPL variables and procedures must be declared before they are used. Another reason is that XPL programs should incorporate the elements of good programming style, such as the use of procedures, literals, generous comments, and general top-down design.

One of the most useful tools in programming is the ability to put comments into source code files. Comments can improve readability and provide important program documentation. An XPL comment is a sequence of characters that begin with the character pair /* and end with the character pair */. These delimiters instruct the compiler to ignore any text between them. A comment can appear anywhere a space character may, thus comments can be freely distributed throughout an XPL program. XPL comments cannot be nested.

Because XPL procedures must be declared before they are used, an XPL program has the general structure of having the low-level procedures first, working to the top-level procedures near the end. The main program code is usually the last thing in the source code file. Global literals and declarations should appear at the beginning of the program, so that they can be used throughout the program and are easily found by the programmer. Block comments should be used to describe the function of each procedure, and individual comments should accompany each non-obvious program statement.

The basic guideline to follow is that an XPL program should be organized so that it is easy to read and understand. Procedures should be grouped according to function, and comments should explain clearly what each section of the program is doing. Declarations should be located where they would logically be looked for by another programmer. Such considerations make understanding and maintaining programs much faster and easier.

Notice the following outline of general XPL program structure.

```
/*  Program Name

    Information here in the comment telling about the program,
    such as:
         - what the program does
         - general algorithms or methods
         - any special constraints or necessary hardware
         - acknowledgements, programmer's name
         - date last modified, who did it, what was done
*/

global literals
global declarations

/* Block comment about the following procedure */

Procedure definition:
   parameter declarations
   local procedure declarations

   procedure statement  /* comment what it does */
   procedure statement  /* comment this one too */
   ...
end procedure;

/* Block comment about the following procedure */

Procedure definition:
   parameter declarations
   local procedure declarations

   procedure code
end procedure;

/* Comment that says MAIN CODE */

main program declarations
main program body
```

Procedures

A procedure is part of a program that is separately defined, and is then invoked from other parts of the program. When a procedure is called, program control is transferred from the point of call to the top of the procedure, the procedure body is executed, and then program control is returned to the point of call.

During the procedure call, parameters (also called arguments) can be specified and passed to the procedure. Each parameter is either an expression or an array. Additionally, each procedure can return a value for use in an arithmetic expression. All these features will be described in the following sections.

A procedure must be defined before it can be used by a program. The basic procedure definition consists of the procedure name and the procedure code body:

```
<proc name>:  procedure;
   <local variable declarations>

   <statement>;
   <statement>;
   ...
end <proc name>;
```

The procedure name is an identifier which will be used in later reference to the procedure. Any legal identifier can be used for the procedure name. In the procedure definition, the procedure name is followed by a colon (:) and the keyword PROCEDURE (which can be shortened to PROC). The statements within the procedure body can be any valid XPL statements, including definitions of other procedures.

A procedure is invoked with the CALL statement:

```
call <proc name>;
```

At the point of the CALL statement, program control passes to the procedure definition. The procedure body is executed, and when it is finished, control is returned to the statement immediately following the CALL statement.

Parameter Passing

Procedures can have one or more parameters. Parameters are values (or arrays) that are passed to the procedure at the point of call for use by the procedure body. For example, if a procedure is defined to output error messages, a special error code could be passed to the procedure when it is called and then the appropriate message would be printed on the terminal.

Parameters must be specified in two places: the CALL statement and the procedure definition. In order to pass parameters to a procedure, the CALL statement must include a parameter list, which is a comma-separated list of parameters enclosed in parentheses:

```
call <proc name> (<act1>, <act2>, ..., <actN>);
```

The parameters ACT1, ACT2, and ACTN are called actual parameters. They are the values actually passed to the procedure, and can be any valid expression or array. Different values or variables can be used as actual parameters each time a procedure is called.

In order for the procedure to properly receive the parameters it is being passed, the procedure definition must also include a parameter list. The parameters in the procedure definition are called the formal parameters, and take the following form:

```
<procname>:  proc (<parm1>, <parm2>, ..., <parmN>);
   dcl <parm1> <type1>;
   dcl <parm2> <type2>;
   ...
   dcl <parmN> <typeN>;
   <local variable declarations>

   <statement>;
   <statement>;
   ...
end <procname>;
```

Each of the formal parameters must appear in the parameter list, and each must be declared within the procedure body, preferably in the order in which they appear and before any other local declarations. When the procedure is called, the value of each of the actual parameters in the CALL statement is assigned to the corresponding formal parameter in the procedure definition. The data types of the actual and formal parameters must be the same.

Formal parameters are declared just as normal program variables with the exception of formal parameters that are arrays. In this case, the size of the array must not be specified, and the keyword ARRAY must be appended to the parameter's data type, as shown in the following example:

```
print_numbers: proc (i, x, list);
   dcl i    fixed;
   dcl x    floating;
   dcl list fixed array;  /* notice special form */

   print 'Integer        = ', i;
   print 'Floating point = ', x;
   print 'Array values   = ', list (0), ' ', list (1);
end print_numbers;

dcl a (9) fixed;
dcl y     floating;
...
call print_numbers (4, y + 1, a);
```

In the example, a procedure is used to print out four numbers in a particular format. Notice that the procedure is passed the correct number of parameters (three), and that each of them is the correct type (fixed, floating, fixed array). A compiler error will occur if the number and types of the parameters do not match.

When the procedure PRINT_NUMBERS is called, the formal parameters I, X, and LIST will take on the values specified in the actual parameter list. The parameter I will be 4, and X and LIST will become the values of Y + 1 and A at the time the procedure is called.

When an array is passed as a parameter, the size of the array is not specified in the DECLARE statement within the procedure body. Since arrays of any size can be passed as procedure parameters, the compiler does not check the limits of subscripts. Care must be taken not to exceed the maximum subscript of a passed array within a procedure body. If a programming error causes a subscript to exceed the amount of storage reserved for an array, random locations of memory will be accessed and perhaps overwritten. Extreme care should be taken to avoid this type of error.

Passing by Value

All scalar variables (i.e., everything except arrays) are passed to procedures by value. As the procedure is called, the current value of the specified variable is copied into a section of memory reserved for the corresponding formal parameter. If, within the procedure body, a new value is assigned to the formal parameter, the value of the actual parameter (the constant or variable that was specified in the CALL statement) will not be changed. For example:

```
put: proc (num);
   dcl num fixed;

   num = 30;
   print num;     /* NUM has been changed to 30 */
end put;

declare a fixed;

a = 10;           /* value of A is 10 */
print a;
call put (a);     /* procedure call... */
print a;          /* A is still 10 */
```

When the program is executed, the variable A is assigned a value of 10 and then printed on the terminal. The PUT procedure is called passing A as a parameter. At this instant, the value of A (which is 10) is copied into a special word of memory reserved for the formal parameter NUM. The PUT procedure is then initiated.

While PUT is being executed, a new value (30) is assigned to NUM. The variable A will not be affected by this, since different memory locations are used for A and NUM. The output of this program is:

```
00010
00030
00010
```

Passing by Reference

Arrays are passed to procedures by reference. Because each array may represent thousands of words of storage, it is impractical to duplicate each array element when a procedure is called. Instead, a pointer to the array is passed to the procedure. If, within the procedure body, a new value is assigned to an element of the array, that element will change in both the formal and actual parameters, since the same location of memory is used for both.

For example:

```
doit: procedure (a);
   dcl a fixed array;

   a (5) = 25;          /* DOIT changes it to 25 */
   print a (5);
end doit;

declare list (10) fixed;

list (5) = 10;          /* this element is 10 */
print list (5);
call doit (list);       /* procedure call...  */
print list (5);         /* and now it is still 25 */
```

The DOIT procedure requires one parameter, a fixed point array. When the fifth element of the formal parameter A is changed, the fifth element of the actual parameter LIST is also changed, since both arrays occupy the same storage area. The output of this program is:

```
00010
00025
00025
```

Type Conversion

The Scientific XPL compiler will automatically perform certain type conversions during a procedure call. No type conversion is required if both the actual and formal parameters are of the same type. If, however, a fixed point quantity is passed to a formal parameter of type floating, the specified quantity is converted to a floating point number before being stored. This conversion results in no loss of data or accuracy.

The compiler, however, cannot automatically convert from a floating point to fixed point quantity without risking the loss of important fractional information. If the user wishes to pass a floating point quantity to a procedure whose corresponding formal parameter is one of type fixed, the INT function must be used to convert the floating point value to fixed point. The fractional part of a floating point quantity is lost in the conversion from floating point to fixed point.

The following program demonstrates the different type conversions that can be performed by the compiler. POWER is a procedure that takes two parameters, the first of type floating and the second of type fixed. The procedure POWER raises the first parameter to the power of the second parameter and prints the floating point result.

```
power: proc (num, pow); /* raise a number to a certain power */
   dcl num    floating; /* number */
   dcl pow    fixed;    /* exponent */
   dcl result floating;
   dcl i      fixed;

   result = 1.0;            /* initialize RESULT */

   do i = 1 to pow;         /* loop POW times */
      result = result*num;  /* raise NUM to the POW power */
   end;

   print result;            /* print the result */
end power;

dcl a fixed;
dcl b floating;

a = 4; b = 3.5;

call power (a, a);          /* fixed, fixed */
call power (a, int (b));    /* fixed, floating */
call power (b, a);          /* floating, fixed */
call power (b, int (b));    /* floating, floating */
```

In the first CALL statement above, POWER is passed two fixed point parameters. The first parameter (A) will be converted to floating point during the procedure call. In the second CALL, however, the INT function must be used to explicitly extract the integer part of the floating variable B for passage to the procedure POWER. Note that the parameter passing rules are identical to the assignment statement conversion rules.

The RETURN Statement

A procedure can return control to the point of call at any time by the use of a RETURN statement. Upon execution of a RETURN statement, a jump back to the point of call is performed, and any remaining lines of the procedure will not be executed. The RETURN statement is often used to check for certain conditions at the beginning of a procedure before it proceeds to execute the code body. For example:

```
print_message: proc (message);
   dcl message fixed array; /* character string */

   if message (0) = 0 then return;  /* string is empty */

   print; print 'Error! ',;
   print string (message);
   print;
end print_message;
```

The above procedure should only print an error if it is passed a non-empty string. The length of the string is therefore checked at the beginning, and if the length is zero, the RETURN statement returns control back to the point of call without executing the rest of the procedure.

Never use a GOTO statement to leave a procedure body. Not only is this a poor programming technique, but information will continue to accumulate on the push down stack. Always use a RETURN statement to leave a procedure. Note that there is an implied return at the end of every procedure.

Functions

Functions are procedures that return a value back to the point of call. Functions can be very useful when computing arithmetic equations, or when error conditions or status codes need to be checked. The type of the value to be returned is specified with the RETURNS attribute of the procedure definition, which appears after the parameter list. The procedure type defines the precision of the value returned so that proper type conversion takes place when the procedure is invoked as part of an arithmetic expression. To return from a function, the RETURN statement followed by the value to return must be used.

A function can simply be called (to ignore the returned value), or a variable can be specified to receive the returned value. Functions can also be used within an expression. The following program is one example of how a function can be used:

```
sum: proc (a, b, c) returns (floating);
   dcl (a, b, c) floating;

   return (a + b + c); /* compute and return answer */
end sum;

dcl (x, y, z) floating;  /* three input numbers */
dcl answer    floating;  /* result of the sum */

do while (true);                          /* loop forever */
   print 'Enter three numbers x,y,z',; /* get three numbers */
   input x, y, z;

   answer = sum (x, y, z);                /* add them up */

   print 'Sum     = ', answer;            /* print the sum */
   print 'Average = ', answer/3;          /* print the average */
   print;
end;
```

The procedure SUM is a function that returns the sum of three numbers. Notice the use of the RETURN statement in the procedure body, and also the attribute RETURNS (FLOATING) in the procedure definition following the parameter list. A function must always end with a RETURN statement. Any RETURN statements within a function must specify the value that is to be returned.

This procedure definition could also be written without the word RETURNS in it, for compatibility with older programs:

```
sum: proc (a, b, c) floating;
```

If the value to be returned is a fixed point number, it is also possible to omit the return type completely. The compiler assumes that if a value is returned by a function and the type is not specified in the procedure definition, the returned value is fixed point. Neither of these older forms are recommended, however, because the RETURNS attribute immediately tells the programmer that a value is returned from that procedure.

A procedure that returns a value can be called as part of a standard arithmetic expression. For example, the variable ANSWER could be eliminated in the above program if the program statements were as follows:

```
print 'Sum     = ', sum (x, y, z);
print 'Average = ', sum (x, y, z)/3;
```

When a procedure is called in this manner, the numeric value returned by the procedure (using the RETURN statement) is used in the expression computation. Arithmetic expressions that call procedures may be used anywhere XPL allows numeric expressions. Multiple or nested procedure calls are also allowed within one expression.

Functions can be used to improve code legibility and detect errors. The following program returns a boolean status to the main program:

```
print_square: proc (x) returns (boolean);
   dcl x floating;

   if x < 0 then return (false);     /* negative number */

   print 'Square root = ', sqr (x);  /* built-in function SQR */
   return (true);                    /* okay, did it */
end print_square;

dcl num floating;

do while (true);                        /* loop forever */
   print 'Number to find square root of',;
   input num;

   if not print_square (num) then do;  /* if FALSE returned */
      print 'Sorry, cannot take the square root of negative';
      print 'numbers.  Please try again.';
   end;
end;
```

The procedure PRINT_SQUARE first checks the parameter it received to make sure it is a positive number. If it is not, the RETURN statement is used to return a false value which makes the logical statement 'IF NOT' in the main program work correctly. Such logical relationships can often be used in XPL programming to increase program readability, or to signal that a procedure encountered an error.

Block Structure and Scope

XPL is a block-structured language. This means that a certain portion of a program, namely a block, can be written so there is no unwanted interaction between the block and its environment. This desirable situation stems from the concept of scope. Entities which are declared within a block are inaccessible to statements or declarations outside the block, and a block can shield itself from the influence of entities declared outside the block by suitable declarations inside the block. The use of the same identifiers for different objects, one inside a block and another outside the block, creates no problem.

For example, there are two blocks in the following program:

```
declare (a, b) fixed;

a = 3;

begin;
  declare c fixed;

  c = a - 17;
end;

b = a + 200;
```

The BEGIN-END group constitutes a block, as does the entire program. The scope of the variables A and B comprises the entire program, because they were declared in the outermost block. The scope of the variable C is within the BEGIN-END block only, because C was declared within that block. This means that the variables A and B may be used anywhere in the program, while the use of the variable C is restricted to the BEGIN-END block. A reference to C located outside the BEGIN-END block would result in an error message from the compiler; outside its scope, the variable C does not exist.

All identifiers are subject to scope. This includes names that represent variables, arrays, procedures, literals, labels.

How Scope is Defined

A block is a BEGIN-END block, any procedure body, or the entire program. Each block limits the scope of those identifiers declared within it; they will be unknown outside the block. Given an identifier, its scope is determined by finding the point of its declaration, and looking forward and backward in the program text ("outward" from the declaration) to find the innermost block containing the declaration. The exact scope of the identifier then begins with its declaration, and ends with the end of that block.

The scope of an identifier, so defined, can have "holes" in it. If the scope contains an inner block, and the inner block contains a declaration that redefines the same identifier, then the scope of that inner declaration creates an area in which the outer declaration is temporarily inoperative, masked by the inner declaration.

Notice the following example:

```
 1   declare (a, b) fixed;
 2
 3   b = 0;
 4
 5   p: proc (a) returns (floating); /* procedure P */
 6      dcl a floating;
 7
 8      return (a*a + b);          /* returns A*A+B */
 9   end p;
10
11   a = p (2);                    /* call P, store result in A */
12
13   begin;                        /* start of block */
14      declare p (10) floating; /* P is now an array */
15      declare i      fixed;
16
17      do i = 0 to 9;             /* loop */
18         p (i) = 500 + i;
19      end;
20
21      a = p (2);                 /* move element of P to variable A */
22   end;                          /* end of block */
```

Consider the scope of the variable I in the above program. I is declared at line 15; the innermost block encompassing this declaration is the BEGIN-END block comprising lines 13 through 22. Therefore, the scope of the variable I begins with its declaration at line 15, and ends with the end of the block at line 22.

The scope of the variable B begins with the declaration at line 1, and ends with the end of the program at line 22. That is to say, the scope of B is the entire program. The case of variable A is similar, since it is declared simultaneously with B, but there is an important difference. The procedure P, whose definition begins at line 5, contains the declaration of another variable A whose scope is the body of the procedure P, lines 5 through 9. So there are two distinct variables named A in this program, declared at two different block levels. The scope of the outer A fails to be continuous; it extends from line 1 to line 4, and from line 10 through line 22. It is interrupted by the scope of the inner A, which occupies lines 5 through 9. Thus the multiplication at line 8 uses the inner A (the formal parameter of procedure P), and the assignment statement at line 11 assigns an initial value to the outer A, the one declared at line 1.

The scope of B is not interrupted by any inner declaration in procedure P. That is why the reference to B at line 8, although within the procedure, is nonetheless a reference to the global B declared at line 1.

Let us now consider the scope of the procedure P. Its definition begins at line 5, and the innermost block encompassing this declaration is the entire program. The scope of procedure P is the entire program (from line 5 down) with one exception. Notice that the identifier P is declared again at line 14, this time not as a procedure, but as a ten element flaoting point array. As in the case of the identifier A, this double declaration presents no difficulty because the declaration at line 14 is contained within an inner block, in this case the BEGIN-END block encompassing lines 13 through 22. Without this inner declaration of the identifier P, the scope of the procedure P would be from line 5 down; with it, the scope of the procedure is only from line 5 through line 13.

The double declaration of P - once as a procedure and once as an array - has a curious consequence. The two statements at lines 11 and 21 (A = P(2)), although lexically identical, have completely different meanings. Line 21 falls within the scope of the array declaration at line 14, and thus sets the variable A equal to the third element of the array P (which the iteration of lines 17 through 19 has left equal to 502). Line 11, on the other hand, falls outside the scope of the array P, and within the scope of procedure P. Thus the assignment of line 11 invokes the procedure P with an actual parameter of 2; within the procedure body the inner variable A becomes equal to 2; the value 2*2 + B, or 4, is returned as the value of the procedure call, and the outer A gets assigned the value of 4.

The Scope of Labels

It is not usually required to explicitly declare label names. The first use of an undeclared label is itself an implicit declaration of the label, and this implicit declaration governs the scope of the label according to the rules mentioned above. But there are times when a programmer must override these implicit declarations with his own explicit declarations.

The form of this type of declaration is:

```
declare <label name> label;
```

Such a declaration specifies that the label will be defined at the block level of the declaration. This explicit label declaration is necessary only if the implied declaration does not satisfy the needs of the programmer.

Here is an example that shows why the explicit declaration is sometimes required:

```
 1   dcl x    fixed;
 2   dcl EXIT label;
 3
 4   x = x + 1;                  /* start of outer block */
 5
 6   begin;                      /* start of inner block */
 7      ...
 8      goto EXIT;
 9      ...
10   end;                        /* end of inner block */
11
12   EXIT: stop;                 /* exit is here */
```

In the above program, the obvious intention is to branch from the inner block (at line 8) to the outer block (at line 12). If line 2 (the label declaration) was not in the program, the label EXIT would first be declared (implicitly) at line 8 (GOTO EXIT). Since the label was implicitly declared at line 8, its scope would span the innermost block from line 8 through line 10. When the compiler encountered the END at line 10, it would try to limit the scope of the label EXIT, only to find that it has not been defined. The explicit declaration at line 2 serves to expand the scope of the label EXIT to encompass lines 2 through 12 (essentially the entire program).

The Use of Block Structure

Transfer of program control from one block nesting level to another should always be done by entering the block at its beginning and leaving at its end, or, for a procedure body, leaving by means of the RETURN statement.

For example, a GOTO statement which contrives to jump into the middle of a procedure body will leave the runtime push down stack in an undefined state, and continued execution of the program will produce unpredictable results. A procedure body must always be entered by means of a call to the procedure. A GOTO statement leaving a procedure body has similar trouble with the runtime stack, since it bypasses the orderly return mechanism. Because of this, it is illegal to write a GOTO statement inside a procedure that transfers control outside of the procedure. In general, GOTO statements from the middle of one block to the middle of another should be avoided at all costs. Programs can often be reorganized to remove the need for such GOTO statements.

It is recommended that within any given block all declarations be put at the beginning of the block, preceding executable statements. The scope of such identifiers can then be visualized as the extent of the entire block. This simplification also prevents an important class of programming errors: mistaken identification of the "innermost encompassing block".

Programmers find their work greatly facilitated by proper layout of a program on the pages of its program listing. Blocks (procedures, DO-groups) are frequently set off by blank lines. The body of each block is indented by a fixed number of spaces (usually three) from the code in which it occurs, thus the opening and closing lines of the block are vertically aligned. When you look at a program listing it should be easy to see the block nesting structure at a glance, without reading the code in detail.

Block structure in a programming language provides the opportunity to define truly independent program modules, letting the compiler do the work of keeping them independent. Procedures can be made independent of their environment (except for the number and type of parameters). Procedures can be moved from one program to another, with no surprise results from the new declaration. Complete self-contained modules, together with conventional literal definitions, can form a project or department library, greatly reducing program development time.

Variable Storage Classes

The storage class of an XPL variable determines its lifetime (when it exists) and its scope (area of influence). There are four storage classes of variables, two of which are meaningful only within the context of procedures: STATIC and AUTOMATIC. Both of these types follow the scope rules explained in the previous section; they differ only in their lifetimes. The other two storage classes (PUBLIC and EXTERNAL) are explained in the next section, "Modules and Libraries".

Static variables exist over the entire lifetime of the program, from the moment the program starts executing until the moment it terminates. They are initialized (to zero) only once, at the start of program execution. Any value that is stored in a static variable is retained until explicitly overwritten by the programmer's code.

Automatic variables, on the other hand, exist only when the block in which they are defined is active (i.e., while that block is being executed). They are created and initialized (to zero) at the start of the block and destroyed when the block is finished executing. Thus they are dynamic, existing only when needed. They can be used to save memory space in a program, although the execution time of the program will necessarily be increased. Automatic variables can be used only in procedures.

The storage class of a variable appears directly after the variable type in the declaration statement. The following examples are all valid variable declarations:

```
dcl i     fixed static;
dcl x     floating automatic;
dcl a (9) boolean automatic;
```

All variables in XPL are static by default (i.e., if no storage class is specified). If a variable must be static in order for the program to work correctly, it is a good idea to explicitly declare it to be static for documentation purposes.

Automatic variables are allocated off the runtime push down stack. It is important to make sure this stack is large enough to avoid a stack overflow, or else erroneous program results will occur. If the default stack is not large enough, its size can be increased with the PDL statement (see the appendix "Compilation Control").

There is one restriction with the use of automatic variables that does not exist for static variables. Automatic variables cannot be accessed by a procedure that is nested within the procedure that declared the variable. If the nested procedure needs to access the variable, the variable must either be static or it must be passed to the procedure as a parameter.

Recursive Procedures

Variables in procedures are static by default. If a procedure's function is recursive in nature, it is possible to make the variables within that procedure default to automatic. This is done in the procedure definition by affixing the keyword RECURSIVE after the returns type:

```
<proc name>: proc (<parm list>) returns (<type>) recursive;
```

The following example is a recursive factorial function:

```
factorial: proc (x) returns (fixed) recursive;
   dcl x fixed; /* automatic by default */

   if x <= 1  /* 0! and 1! are both 1 */
   then return (1);
   else return (x*factorial (x - 1));  /*   x*(x - 1)!  */
end factorial;
```

Any local variable declared in a recursive procedure that needs to be static must explicitly be declared to be static. Note that recursive procedure formal parameters cannot be declared static; they must be automatic. If a procedure is defined within a recursive procedure, it is recursive by default and cannot be changed.

Recursive procedures quickly use up the runtime stack. See the warning about automatic variables above.